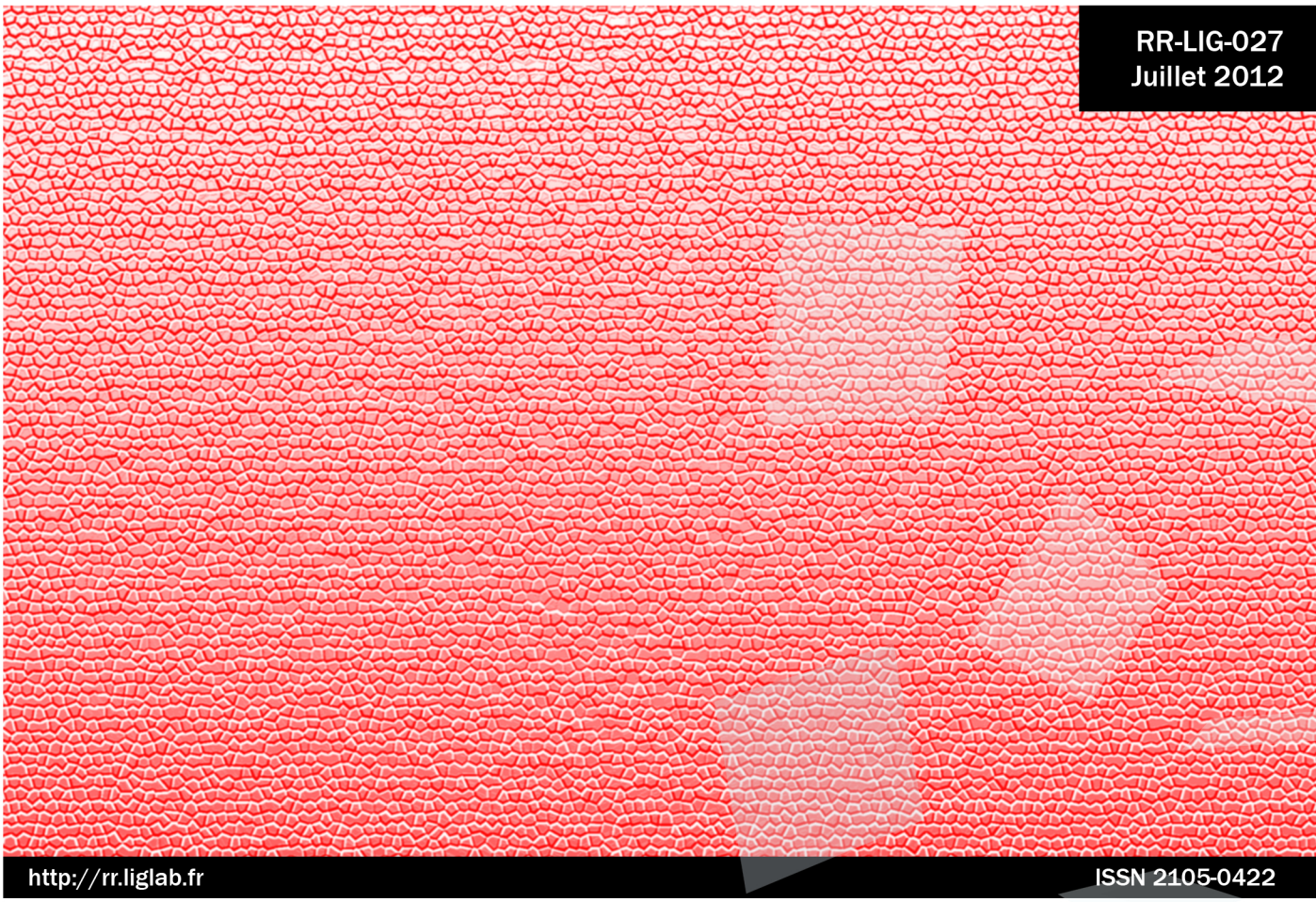
Les rapports de recherche du LIG

Benchmarking Dependability of Map Reduce Systems

Amit SANGROYA, PhD student, LIG, Inria, Grenoble University, France
Damián SERRANO, Post-doc, LIG, Inria, France
Sara BOUCHENAK, Associate Professor, Grenoble University (UJF), France



RR-LIG-027
Juillet 2012

http://rr.liglab.fr
ISSN 2105-0422

L I G

Benchmarking Dependability of MapReduce Systems

Amit Sangroya
INRIA - LIG
Grenoble, France
Amit.Sangroya@inria.fr

Damián Serrano
INRIA - LIG
Grenoble, France
Damian.Serrano@inria.fr

Sara Bouchenak
University of Grenoble - LIG - INRIA
Grenoble, France
Sara.Bouchenak@inria.fr

***Abstract*—MapReduce is a popular programming model for distributed data processing. Extensive research has been conducted on the reliability of MapReduce, ranging from adaptive and on-demand fault-tolerance to new fault-tolerance models. However, realistic benchmarks are still missing to analyze and compare the effectiveness of these proposals. To date, most MapReduce fault-tolerance solutions have been evaluated using microbenchmarks in an ad-hoc and overly simplified setting, which may not be representative of real-world applications. This paper presents MRBS, a comprehensive benchmark suite for evaluating the dependability of MapReduce systems. MRBS includes five benchmarks covering several application domains and a wide range of execution scenarios such as data-intensive vs. compute-intensive applications, or batch applications vs. online interactive applications. MRBS allows to inject various types of faults at different rates. It also considers different application workloads and dataloads, and produces extensive reliability, availability and performance statistics. We illustrate the use of MRBS with Hadoop clusters running on Amazon EC2, and on a private cloud.**

***Keywords*-Benchmark; Dependability; MapReduce; Hadoop**

I. Introduction

MapReduce has become a popular programming model and runtime environment for developing and executing distributed data-intensive and compute-intensive applications [1]. It offers developers a means to transparently handle data partitioning, replication, task scheduling and fault-tolerance on a cluster of commodity computers. Hadoop, one of the most popular MapReduce frameworks, provides key fault-tolerance features such as handling node failures and task failures.

There has been a large amount of work towards improving fault-tolerance solutions in MapReduce. Several efforts have explored on-demand fault-tolerance [2], replication and partitioning policies [3], [4], adaptive fault-tolerance [5], [6], and extending MapReduce with other fault-tolerance models [7], [8]. However, there has been very little in the way of empirical evaluation of MapReduce dependability. Evaluations have often been conducted in an ad-hoc manner, such as turning off a node in the MapReduce cluster or killing a task process. These actions are typically dictated by what testers can actually control, but may lead to low coverage testing. Recent tools, like Hadoop fault injection framework [9], offer the ability to emulate non-deterministic exceptions in the HDFS distributed filesystem underlying Hadoop MapReduce. Although they provide a means to program unit tests for HDFS, such low-level tools are meant to be used by developers who are familiar with the internals of HDFS, and are unlikely to be used by end-users of MapReduce systems. MapReduce fault injection must therefore be generalized and automated for higher-level and easier use.

Not only it is necessary to automate the injection of faults, but also the definition and generation of MapReduce faultloads. A faultload will describe *what* fault to inject (e.g. a node crash), *where* to inject it (e.g. which node of the MapReduce cluster), and *when* to inject it (e.g. five minutes after the application started). Furthermore, most evaluations of MapReduce fault-tolerance systems relied on microbenchmarks based on simple MapReduce programs and workloads, such as *grep*, *sort* or *word count*. While microbenchmarks may be useful in targeting specific system features, they are not representative of full distributed applications, and they do not provide multi-user realistic workloads.

In this paper, we present MRBS (MapReduce Benchmark Suite), the first benchmark suite for evaluating the dependability of MapReduce systems. MRBS enables automatic faultload generation and injection in MapReduce. This covers different fault types, injected at different rates, which will provide a means to analyze the effectiveness of fault-tolerance in a variety of scenarios. MRBS allows to quantify dependability levels provided by MapReduce fault-tolerance systems, through an empiric evaluation of the availability and reliability of such systems, in addition to performance and cost metrics. In addition, MRBS covers five application domains: recommendation systems, business intelligence, bioinformatics, text processing, and data mining. It supports a variety of workload and dataload characteristics, ranging from compute-oriented to data-oriented applications, batch applications to online interactive applications. Indeed, while MapReduce frameworks were originally limited to offline batch applications, recent works are exploring the extension of MapReduce beyond batch processing [10], [11]. MRBS uses various input data sets from real applications, among which an online movie recommender service [12], Wikipedia [13], and real genomes for DNA sequencing [14].

We illustrate the use of MRBS to evaluate Hadoop fault-tolerance capabilities, when running the Hadoop cluster on Amazon EC2, and on a private cloud. MRBS shows that when running the Bioinformatics workload and injecting a faultload that consists of a hundred map software faults and three node faults, Hadoop MapReduce handles these failures with high reliability (94% of successful requests) and high availability (96% of the time). MRBS can also be used for other purposes, such as the evaluation and comparison of different MapReduce fault-tolerance approaches, or choosing the right MapReduce cluster configuration to meet dependability and performance objectives. In this paper, we illustrate two use-cases of MRBS.

We wish to make dependability benchmarking easy to adopt by end-users of MapReduce and developers of MapReduce fault-tolerance systems. MRBS allows automatic deployment of experiments on cloud infrastructures. It does not depend on any particular infrastructure and can run on different private or public clouds. MRBS is available as a software framework to help researchers and practitioners to better analyze and evaluate the dependability and performance of MapReduce systems.

The remainder of the paper is organized as follows. Section II describes the background on MapReduce. Sections III-IV describe the dependability benchmarking in MRBS. Section V presents experimental results, and Section VI discusses use cases of MRBS. Section VII reviews the related work, and Section VIII draws our conclusions.

II. System and Programming Model

MapReduce is a programming model and a software framework to support distributed computing and large data processing on clusters of commodity machines [1]. High performance and fault-tolerance are two key features of MapReduce. They are achieved by automatic task scheduling in MapReduce clusters, automatic data placement, partitioning and replication, and automatic failure detection and task re-execution. A MapReduce *job*, i.e. an instance of a running MapReduce program, has several phases; each phase consists of multiple *tasks* scheduled by the MapReduce framework to run in parallel on cluster nodes. First, input data are divided into *splits*, one split is assigned to each map task. During the mapping phase, tasks execute a *map* function to process the assigned splits and generate intermediate output data. Then, the reducing phase runs tasks that execute a *reduce* function to process intermediate data and produce the output.

A. Hadoop MapReduce

There are many implementations of MapReduce. Hadoop is a popular MapReduce framework, available in public clouds such as Amazon EC2, and Open Cirrus. A Hadoop cluster consists of a *master node* and *slave nodes*. Users (i.e. clients) of a Hadoop cluster submit MapReduce jobs to the master node which hosts the *JobTracker* daemon that is responsible of scheduling the jobs. By default, jobs are scheduled in FIFO mode and each job uses the whole cluster until the job completes. However, other multi-user job scheduling approaches are also available in Hadoop to allow jobs to run concurrently on the same cluster. This is the case of the *fair scheduler* which assigns every job a fair share of the cluster capacity over time. Moreover, each slave node hosts a *TaskTracker* daemon that periodically communicates with the master node to indicate whether the slave is ready to run new tasks. If it is, the master schedules appropriate tasks on the slave. Each task is executed by a separate process.

Hadoop framework also provides a distributed filesystem (HDFS) that stores data across cluster nodes. HDFS architecture consists of a *NameNode* and *DataNodes*. The *NameNode* daemon runs on the master node and is responsible of managing the filesystem namespace and regulating access to files. A *DataNode* daemon runs on a slave node and is responsible of managing storage attached to that node. HDFS is thus a means to store input, intermediate and output data of Hadoop MapReduce jobs. Furthermore, for fault tolerance purposes, HDFS replicates data on different nodes.

B. Failures in Hadoop MapReduce

One of the major features of Hadoop MapReduce is its ability to tolerate failures of different types [15], as described in the following.

Node Crash: In case of a slave node failure, the JobTracker on the master node stops receiving heartbeats from the TaskTracker on the slave for an interval of time. When it notices the failure of a slave node, the master removes the node from its pool and reschedules tasks that were ongoing on other nodes. The heartbeat timeout is set in the *mapred.task.tracker.expiry.interval* Hadoop property. In the current implementation of Hadoop, failures of the master node are not tolerated.

Task Process Crash: A task may also fail because a map or reduce task process suddenly crashes, e.g., due to a transient bug in the underlying (virtual) machine. Here again, the parent TaskTracker notices that a task process has exited and notifies the JobTracker for possible task retries.

Task Software Fault: A task may fail due to errors and runtime exceptions in *map* or *reduce* functions written by the programmer. When a TaskTracker on a slave node notices that a task it hosts has failed, it notifies the JobTracker on the master node which reschedules another execution of the task, up to a maximum number of retries. This allows to tolerate transient errors in MapReduce programs.

Hanging Tasks: A map or reduce task is marked as failed if it stops sending progress updates to its parent TaskTracker for a period of time (indicated by *mapred.task.timeout* Hadoop property). If that occurs, the task process is killed, and the JobTracker is notified for possible task retries.

III. Dependability Benchmarking in MRBS

To use MRBS, three main steps are needed: (i) build a faultload (i.e. fault scenario) to describe the set of faults to be injected, (ii) conduct fault injection experiments based on the faultload, and (iii) collect statistics about dependability levels of the MapReduce system under test. This is presented in Figure 1.

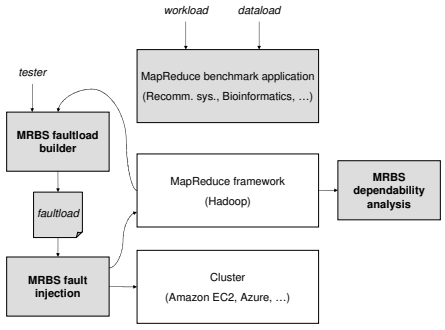


Figure 1. Overview of MRBS dependability benchmarking

The evaluator of the dependability of a MapReduce system chooses an application from MRBS' set of benchmarks, depending on the desired application domain and whether he/she targets compute-oriented or data-oriented applications. MRBS injects (default) workload and dataload in the system under test. MRBS also allows the evaluator to choose specific dataload and workload, to stress the scalability of the MapReduce system. Further details about workload and dataload injection, and dependability analysis in MRBS are given in Section IV.

MRBS is available as a software framework to help testers to better analyze and evaluate the dependability of MapReduce systems. The current implementation of MRBS runs on top of Hadoop MapReduce [15]. It can be downloaded from *http://sardes.inrialpes.fr/research/mrbs/*

A. Faultload Builder

A faultload in MRBS is described in a file, either by extension, or by intention. In the former case, each line of the faultload file consists of the following elements: the time at which a fault occurs (relatively to the beginning of the experiment), the type of fault that occurs and, optionally, where the fault occurs. A fault belongs to one of the fault types handled by Hadoop MapReduce[1], and introduced in Section II-B. A fault occurs in one of the MapReduce cluster nodes; this node may be either explicitly specified in the faultload or randomly chosen among the set of nodes. To make the parsing of this faultload file more efficient, redundant lines, that correspond to multiple occurrences of the same fault at the same time, are grouped into one line with an extra parameter that represents the number of occurrences of that fault. Another way to define a more concise faultload is to describe it by intention. Here, each line of the faultload file consists of: a fault type, and the mean time between failures (MTBF) of that type.

Thus, testers can explicitly build synthetic faultloads representing various fault scenarios. A faultload description may also be automatically obtained, either randomly or based on previous application runs' traces. The random faultload builder produces a faultload description where, with each fault type, is associated a random MTBF between 0 and the length of the experiment. Similarly, the random faultload builder may produce a faultload by extention, where it generates the i-th line of the faultload file as follows: $< time\_stamp_i,\ fault\_type_i,\ fault\_location_i >$, with $time\_stamp_i$ being a random value between $time\_stamp_{i-1}$ (or 0 if $i = 1$) and the length of the experiment, $fault\_type_i$ and $fault\_location_i$ random values in the set of possible values. A faultload description may also be automatically generated based on traces of previous runs of MapReduce applications and workloads. The trace-based faultload builder parses the MapReduce framework's logs and identifies the faults that occurred in these runs: their time stamp, their type, and possibly their location. We designed the trace-based faultload builder to work directly on the MapReduce framework's logs, which allows it to work on workloads and benchmark applications from the MRBS benchmark suite, but also with other workloads and MapReduce applications. As with the other variants of the faultload builder, the faultload that results from the trace-based faultload builder may be described by extension or intention. In the latter case, a statistical analysis of the traces is performed to calculate MTBF for the different types of faults.

MRBS faultload builder is relatively portable: its two first variants – explicit faultload builder and random faultload builder – are general enough and do not rely on any specific platform. The trace-based faultload builder is independent from the internals of the MapReduce framework, and produces a faultload description based on the structure of the MapReduce framework's logs; it currently works on Hadoop MapReduce framework.

B. Fault Injection

The output of the MRBS faultload builder is passed to the MRBS fault injector. The MRBS fault injector divides the input faultload into subsets of faultloads: one crash faultload groups all crash faults that will occur in all nodes of the MapReduce cluster (i.e. node crash, task process crash), and one per-node faultload groups all occurrences of other types of faults that will occur in one node (i.e. task software faults, hanging tasks).

The MRBS fault injector runs a daemon that is responsible of injecting the crash faultload. In the following, we present how the daemon injects these faults, in case of a faultload described by extension, although this can be easily generalized

[1]Other types of faults, such as network disconnection, may be emulated by MRBS, although we do not detail them in this paper.

to a faultload described by intention. Thus, for the i-th fault in the crash faultload, the dameon waits until $time\_stamp_i$ is reached, then calls the fault injector of $fault\_type_i$ (see below), on the MapReduce cluster node corresponding to $fault\_location_i$. This fault injector is called as many times as there are occurrences of the same fault at the same time. The fault injection dameon repeats these operations for the following crash faults, until the end of the faultload file is encountered or the end of the experiment is reached.

The MRBS fault injector handles the per-node faultloads differently. First, it synthesizes a new version of the MapReduce framework library using aspect-oriented techniques. The synthetic MapReduce library has the same API as the original one, but underneath this new library includes interceptors that encode the fault injection logic. These interceptors create a fault-oracle per MapReduce node, this oracle is responsible of analyzing the per-node faultload and orchestrating its injection in the tasks running on that node. In addition, task creation is intercepted to automatically ask the fault-oracle whether a fault must be injected in that task, in which case the fault injector corresponding to the fault type is called, as described in the following. The overall architecture of the faultload injection in MRBS is described in Figure 2.

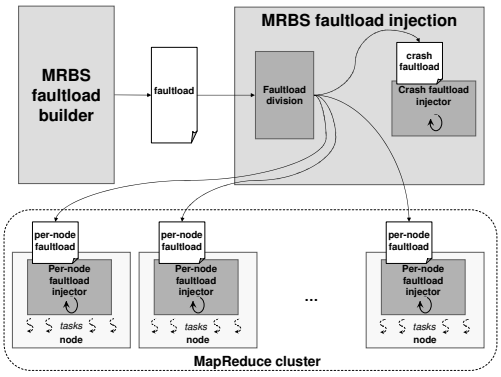


Figure 2. Architecture of the faultload injector

Node Crash Injection: A node crash is simply implemented by shutting down a node. This fault injector uses the API of the underlying cloud infrastructure to implement such a fault. For example, in case of a public cloud such as Amazon EC2, a node crash consists in a premature termination of an Amazon EC2 instance. However, if a tester wants to conduct multiple runs of the same dependability experiment, and if faults are implemented by shutting down machines, new machines must be acquired from the cloud at the beginning of each run, which may induce a delay. For efficiency purposes, we propose an implementation of MapReduce node fault which kills all MapReduce daemons running on that node. Specifically, in the case of Hadoop these include the *TaskTracker* and *DataNode* daemons running in a slave node[2]. The timeout to detect a MapReduce node failure is set to 30 seconds, a value set in *mapred.task.tracker.expiry.interval* Hadoop property.

Task Process Crash Injection: This type of fault is implemented by killing the process running a task on a MapReduce node.

Task Software Fault Injection: A task software fault is implemented as a runtime exception thrown by a map task or a reduce task. This fault injector is called by the interceptors injected into the MapReduce framework library by MRBS.

Provoking Hanging Tasks: A task is marked as hanging if it stops sending progress updates for a period of time. This type of fault is injected into a map task or a reduce task through the interceptors that make the task sleep a longer time than the maximum period of time for sending progress updates (*mapred.task.timeout* Hadoop property).

The MRBS faultload injector is relatively portable: it is independent from the internals of the MapReduce framework and the per-node faultload injectors are automatically integrated within the framework based upon its API. The current version of the MRBS faultload injector works for Hadoop MapReduce; porting to new platforms is straightforward.

IV. Load Injection and Dependability Analysis

MRBS allows to inject various faultloads, workloads and dataloads in MapReduce systems, and to collect information that helps testers understand the behavior observed as a result of fault injection. In addition, MRBS allows to automatically deploy extensive experiments and test various scenarios on cloud infrastructures such as Amazon EC2 and private clouds. The overall architecture of load injection and dependability analysis in MRBS is presented in Figure 3, and is detailed in the following.

A. Benchmark Suite

The MRBS benchmark suite is a set of five benchmarks covering various application domains: recommendation systems, business intelligence, bioinformatics, text processing, and data mining. As we showed in a previous study [16], these benchmarks were chosen to exhibit different behaviors in terms of computation pattern and data access pattern: the recommendation system is a compute-intensive benchmark, the business intelligence system is a data-intensive benchmark, and the other benchmarks are relatively less compute/dataintensive. Conceptually, each benchmark implements a service providing different types of requests, which are issued by external clients. Each client request requires executing one or a series of MapReduce jobs. MRBS allows to emulate clients implemented as external entities that remotely request the service; and the service runs on a MapReduce cluster. The benchmarks are briefly described in the following, more details can be found in [16].

[2]A node crash is not injected to the MapReduce master node since this node is not fault-tolerant, c.f. Section II.

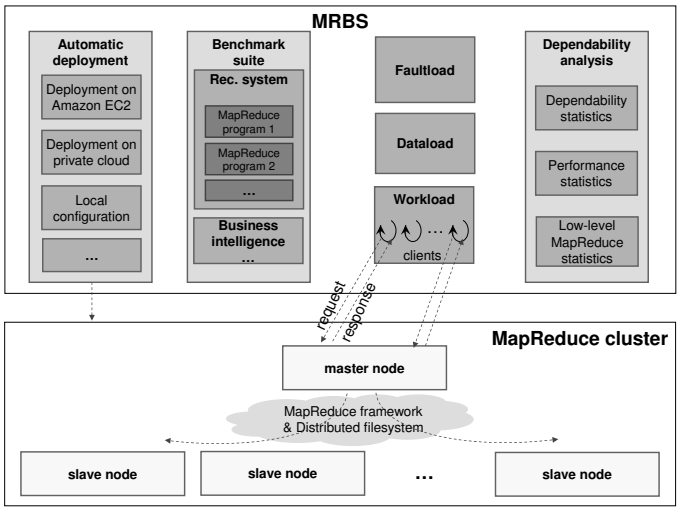


Figure 3. Overview of MRBS

Recommendation System: MRBS implements an online MapReduce-based movie recommender system. It builds upon a set of movies, a set of users, and a set of ratings and reviews users give for movies to indicate whether and how much they liked or disliked the movies. These data have been collected from a real movie recommendation web site [12]. The benchmark provides four types of client requests: (i) provide the top-10 recommendations for a user, (ii) list all the ratings given to a movie, (iii) list all the ratings given by a user, (iv) provide a recommendation for a movie to a given user.

Business Intelligence: This benchmark represents a decision support system for a wholesale supplier, compatible with the TPC-H industry-standard benchmark [17]. It uses Apache Hive on top of Hadoop MapReduce, a data warehouse that facilitates ad-hoc queries using the HiveQL language [18]. The benchmark provides 22 types of client requests that examine large volumes of data, and execute queries with a high degree of complexity, e.g. retrieving the ten unshipped orders with the highest value, or identifying geographies where there are customers who may be likely to make a purchase. The benchmark consists of eight data tables populated with the TPC DBGen TPC package [17], which allows to use input datasets of (almost) any size.

Bioinformatics: Clients of this online bioinformatics service may choose a complete genome to analyze among a set of genomes. The service includes a MapReduce-based implementation of DNA sequencing [19]. DNA sequencing attempts to find where reference reads (i.e. short DNA sequences) occur in a genome. The data used in the benchmark are publicly available genomes [14]. Currently, the benchmark provides three types of client requests.

Text Processing: It is a MapReduce text processing-oriented benchmark, with three types of requests allowing clients to search words or word patterns in text documents, to know how often words occur in text documents, or to sort the contents of documents. These are classical operations for web search engines and log analysis. The benchmark uses Wikipedia dumps of different sizes as its input data [13].

Data Mining: This benchmark provides two types of client requests: classification and clustering [20]. It considers the case of classifying newsgroup documents into categories, and uses collections of data publicly available from [21]. The benchmark also provides canopy clustering operations, which partitions a large number of elements into clusters in such a way that elements belonging to the same cluster share some similarity. The benchmark uses datasets of synthetically generated control charts, to cluster the charts into different classes based on their characteristics [20].

B. Dataload

Interestingly, MRBS considers different aspects of load: faultload, dataload, and workload. This allows testers and developers to stress different aspects of MapReduce systems, such as dependability, and scalability. The dataload is characterized by the size and nature of data sets used as inputs for a benchmark. Obviously, the nature and format of data depend on the actual benchmark and its associated MapReduce programs. For instance, the MRBS movie recommendation system benchmark takes input data consisting of users, movies, and ratings users give to movies. Whereas the bioinformatics benchmark uses input data in the form of genomes for DNA sequencing. Users of MRBS may choose between datasets of different sizes (see Table I, the default input data for each benchmark being the first one).

Table I
MRBS BENCHMARKS' DATALOADS.

Benchmark	Dataload	
Recommendation system	dataload	100,000 ratings, 1000 users, 1700 movies
	dataload+	1 million ratings, 6000 users, 4000 movies
	dataload++	10 million ratings, 72,000 , 10,000 movies
Business intelligence	dataload	1GB
	dataload+	10GB
	dataload++	100GB
Bioinformatics	dataload	genomes of 2,000,000 to 3,000,000 DNA characters
Text processing	dataload	text files (1GB)
	dataload+	text files (10GB)
	dataload++	text files (100GB)
Data mining	dataload	5000 documents, 5 newsgroups, 600 control charts
	dataload+	10,000 documents, 10 newsgroups, 1200 control charts
	dataload++	20,000 documents, 20 newsgroups, 2400 control charts

C. Workload

The workload is characterized by the benchmark to execute, and the number of concurrent clients issuing requests on that benchmark application; this number may vary. The workload is also characterized by the execution mode which may be interactive or batch. In interactive mode, concurrent clients share the MapReduce cluster (i.e. have their requests executed) at the same time. In batch mode, requests from different clients are executed in FIFO order, without concurrently accessing the MapReduce cluster (see Section II-A). Thus, in interactive mode, a client interacts

with the MapReduce service in a closed loop where he requests an operation, waits for the request to be processed, receives a response, waits a think-time, before requesting another operation.

The workload is also characterized by client request distribution, that is the relative frequencies of different request types. It may follow different distribution laws (known as workload mixes), such as a random distribution. Request distribution may be defined using a state-transition matrix that gives the probability of transitioning from one request type to another.

D. Automatic Deployment of Experiments

MRBS enables the automatic deployment of experiments on a cluster in a cloud infrastructure. The cloud infrastructure and the size of the cluster are configuration parameters of MRBS. MRBS acquires on-demand resources provided by cloud computing infrastructures such as private clouds, or the Amazon EC2 public cloud [22]: one node is dedicated to run MRBS load injectors, and the other nodes are used to host the MapReduce cluster. MRBS automatically releases the resources when the benchmark terminates. We expect to provide MRBS versions for other cloud infrastructures such as the OpenStack open source cloud infrastructure [23]. Once the cluster is set up, the MapReduce framework and its underlying distributed file system are started on the cluster. The current implementation of MRBS uses the popular Apache Hadoop MapReduce framework and HDFS distributed file system.

E. Using MRBS

Once the user of MRBS has defined a workload, a dataload, and a faultload (or used the default ones), MRBS automatically deploys the experiment on a cluster in a cloud, and injects the load into the MapReduce cluster. It first uploads input data in the MapReduce distributed file system. This is done once, at the beginning of the benchmark, and the data are then shared by all client requests. Afterwards, it creates as many threads as concurrent clients there are. Thread clients will remotely send requests to the master node of the MapReduce cluster which schedules MapReduce jobs in the cluster (see Figure 3). Clients continuously send requests/receive responses until the execution run terminates. An experiment run has three successive phases: a warm-up phase, a run-time phase, and a slow-down phase. Statistics are produced during the run-time phase, whereas the warm-up phase allows the MapReduce system to reach a steady state before collecting statistics, and the slow-down phase allows to terminate the benchmark in a clean way. An experiment may also be automatically run a number of times, to produce variance reports and average statistics.

To make MRBS flexible, a configuration file is provided, that involves several parameters such as the length of the experiment, the size of MapReduce input data, etc. Nevertheless, to keep MRBS simple to use, these parameters come with default values that may be adjusted by MRBS users.

F. Dependability Analysis

MRBS produces runtime statistics related to dependability, such as reliability, and availability [24]. *Reliability* is measured as the ratio of successful MapReduce client requests to the total number of requests, during a period of time. *Availability* is measured from the client's perspective as the ratio of, on the one hand, the time the benchmark service is capable of returning successful responses to the client, and on the other hand, the total time; availability is measured during a period of time. In addition, MRBS produces performance and cost statistics, such as client request response time, request throughput, and the financial cost of a client request.

MRBS also provides low-level MapReduce statistics related to the number, length and status (i.e. success or failure) of MapReduce jobs, map tasks, reduce tasks, the size of data read from or written to the distributed file system, etc. These low-level statistics are built offline, after the execution of the benchmark. Optionally, MRBS can generate charts plotting continuous-time results.

V. Evaluation

A. Experimental Setup

The experiments presented in Sections V-B and VI were conducted in a cluster running on Amazon EC2 [22], and on two clusters running in Grid'5000 [25]. Each cluster consists of one node hosting MRBS and emulating concurrent clients, and a set of nodes hosting the MapReduce cluster. The experiments below use several benchmarks of MRBS with default dataloads (see Table I); the benchmarks are run in interactive mode, with multiple concurrent clients. In these experiments, client request distribution is random, and request interarrival time is an average of 7 seconds. Availability and reliability are measured in periods of 30 minutes, and cost is based on Amazon EC2 pricing at the time we conducted the experiments, which is \$0.34 per instance·hour. In the following, each experiment is run three times to report average and standard deviation results.

The underlying software configuration is as follows. We used Amazon EC2 large instances which run Fedora Linux 8 with kernel v2.6.21. Nodes in Grid'5000 run Debian Linux 6 with kernel v2.6.32. The MapReduce framework is Apache Hadoop v0.20.2, and Hive v0.7, on Java 6. MRBS uses Apache Mahout v0.6 data mining library [20], and CloudBurst v1.1.0 DNA sequencing library [19]. The hardware configuration used in the experiments is described in Table II.

Table II
HARDWARE CONFIGURATIONS OF MAPREDUCE CLUSTERS

Cluster	CPU	Memory	Storage	Network
Amazon EC2	4 EC2 Compute Units in 2 virtual cores	7.5 GB	850 MB	10 Gbit Ethernet
G5K I	4-core 2-cpu 2.5 GHz Intel Xeon E5420 QC	8 GB	160 GB SATA	1 Gbit Ethernet
G5K II	4-core 1-cpu 2.53 GHz Intel Xeon X3440	16 GB	3200 GB SATA II	Infiniband 20G

B. Experimental Results

In this section, we illustrate the use of MRBS to evaluate the fault-tolerance of Hadoop MapReduce. Here, a ten-node Hadoop cluster runs the Bioinformatics benchmark, used by 20 concurrent clients, in the G5K I cluster. The experiment is conducted during a run-time phase of 60 minutes, after a warm-up phase of 15 minutes. We consider a synthetic faultload that consists of software faults and hardware faults as follows: first, 100 map task software faults are injected 5 minutes after the beginning of the run-time phase, and then, 3 node crashes are injected 25 minutes later. Although the injected faultload is aggressive, the Hadoop cluster remains available 96% of the time, and is able to successfully handle 94% of client requests (see Table III). This has an impact on the request cost which is 14% higher than the cost obtained with the baseline (non-faulty) system.

Table III
RELIABILITY, AVAILABILITY, AND COST

Reliability	Availability	Cost (dollars/request)
94%	96%	0.008 (+14%)

To better explain the behavior of the MapReduce cluster, we will analyze MapReduce statistics, as presented in Figures 4 and 5. Figure 4 presents successful MapReduce jobs and failed MapReduce jobs over time. Note the logarithmic scale of the right side y-axis. When software faults occur, few jobs actually fail. On the contrary, node crashes are more damaging and induce a higher number of job failures, with a drop of the throughput of successfull jobs from 16 jobs/minute before node failures to 5 jobs/minute after node failures.

Figure 5 shows the number of successful MapReduce tasks and the number of failed tasks over time, differenting between tasks that fail because they are unable to access data from the underlying filesystem (i.e. I/O failures in the Figure), and tasks that fail because of runtime errors in all task retries[3] (i.e. task failures in the Figure). We notice that software faults induce task failures that appear at the time the software faults occur, whereas node crashes induce I/O failures that last fifteen minutes after the occurrence of node faults. Actually, when some cluster nodes fail, Hadoop must reconstruct the state of the filesystem, by re-replicating the data blocks that were on the failed nodes from replicas in other nodes of the cluster[4]. This explains the delay during which I/O failures are observed.

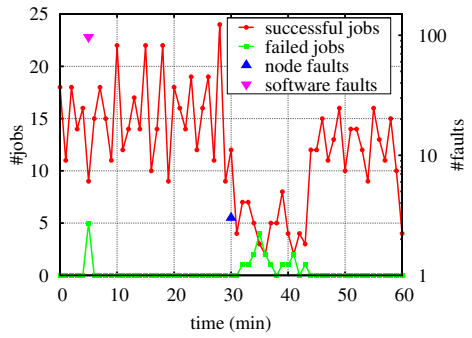


Figure 4. Successful vs. failed MapReduce jobs

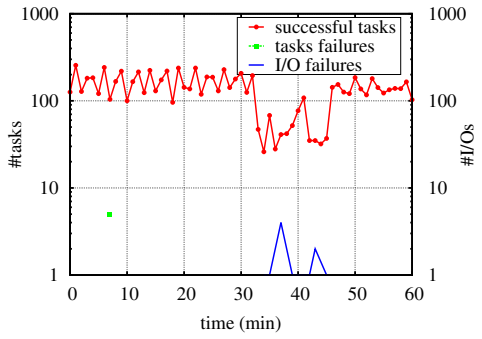


Figure 5. Successful vs. failed MapReduce tasks

We now analyze the impact of these failures on the performance of the Hadoop MapReduce cluster. Figure 6 shows the response time of successful client requests. With software faults, there is no noticeable impact on response times. Conversely, response time sharply increases when there are node faults, and while Hadoop is rebuilding missing data replicas. Similarly, Figure 7 presents the impact of failures on client request throughput. Interestingly, when the Hadoop cluster looses 3 nodes, it is able to fail-over, however, at the expense of a higher response time (+30%) and a lower throughput (-12%).

VI. USE CASES

MRBS has several possible uses, among which helping developers and testers to better analyze the fault-tolerance of MapReduce systems, or to better choose the configuration of the MapReduce cluster to provide service level guarantees. In the following, we present two possible use cases of MRBS.

[3]By default, in Hadoop MapReduce, a task is executed at most four times before it fails.

[4]By default, data have three replicas in HDFS

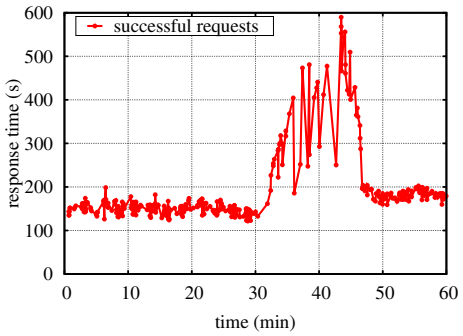


Figure 6. Client request response time

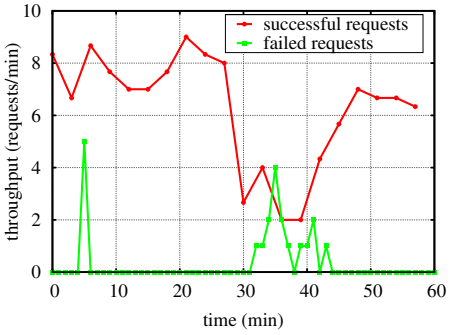


Figure 7. Client request throughput

A. Which MapReduce Cluster Size

We consider the case of a service provider that hosts a data-intensive MapReduce service, such as the Business Intelligence service provided in MRBS [16]. The service provider has to decide the right Hadoop MapReduce cluster size to meet a set of desired service level objectives, combining dependability, performance and cost requirements. The question to answer would have the following form: *What is the MapReduce cluster size that can handle three node failures, while guaranteeing at least 85% of successful requests, and at least 150 requests/hour, with a minimum cost?*

We conducted experiments with MRBS' Business Intelligence service running on Hadoop MapReduce, hosted on the G5K II cluster. We considered an average number of 5 concurrent clients accessing the service, and an input dataload of 1GB, although these values could be changed. We conducted experiments with clusters of various sizes. Each experiment was run three times, and consists of a 30 minute run-time phase, after a 15 minute warm-up phase. For each cluster size, we injected three node faults into the MapReduce cluster at the beginning of the run-time phase.

We report on the measured reliability, throughput and cost in Figure 8; these numbers are average values of three runs, with a relative standard deviation lower than 6% for reliability, and lower than 9% for throughput and cost. Figure 8(a) shows that the MapReduce cluster should include at least 15 nodes to achieve a reliability of 85%. However, for a throughput of at least 150 requests/hour, the MapReduce cluster should have at least 17 nodes, as shown in Figure 8(b). Finally, to meet all these service level objectives with a minimal cost, the MapReduce cluster size should contain 17 nodes (see Figure 8(c)).

B. How Many Faults Are Tolerated?

We consider the case of a service provider that hosts MapReduce services on a ten-node cluster. One question that it has to answer would have the following form: *Up to how many node failures can the MapReduce cluster tolerate, while guaranteeing an availability of at least 85%?*

We conducted experiments with three benchmarks of MRBS, showing three different behaviors: the Business Intelligence service is data-intensive, the Recommendation System is compute-oriented, and the Bioinformatics service is in between (refer to [16] for more details about benchmark profiles). Each benchmark service runs on Hadoop MapReduce, hosted on Amazon EC2. We considered an average number of 5 concurrent clients accessing the service, and the default input dataload, although these values could be changed. We conducted experiments when injecting various numbers of node faults into the Hadoop MapReduce cluster, at the beginning of the run-time phase. Each experiment was run three times, and consists of a 30 minute run-time phase, after a 15 minute warm-up phase.

Figure 9 shows the measured availability, with different faultloads. To guarantee the target availability objective of 85%, the MapReduce cluster hosting the data-intensive Business Intelligence service would not tolerate more than two node failures. In comparison, the less data-intensive Bioinformatics service would tolerate four node failures for the same availability objective, while the compute-oriented Recommendation System would be able to tolerate up to 6 node faults in a ten-node cluster. In summary, Hadoop is able to transparently tolerate failures when there is one node crash. With more node failures, Hadoop MapReduce may handle failures with an acceptable availability level if the MapReduce service it hosts is more compute-intensive than data-intensive.

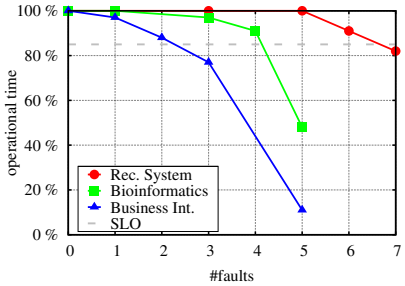
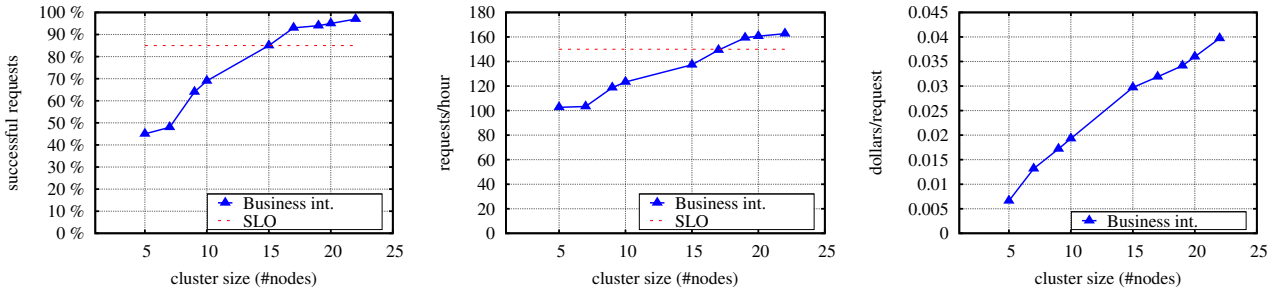


Figure 9. Availability under different node faultloads



(a) Reliability (b) Throughput (c) Cost

Figure 8. Service level objectives to meet

VII. Related Work

Benchmarking is an important issue for evaluating distributed systems. Various research and industry standard performance benchmarking solutions exist such as TPC-C that evaluates on-line transaction processing (OLTP) systems [26], TPC-H for benchmarking decision support systems [17], and YSCB for evaluating different data storage systems in cloud environments [27]. Benchmarks have been also developed for evaluating dependability of distributed systems, such as a benchmark for software RAID Systems [28], a benchmark for OLTP systems [29], and a benchmark for web servers [30].

The need of MapReduce dependability benchmarking is motivated by the many recent works that have been devoted to the study and improvement of fault-tolerance of MapReduce. These include on-demand fault-tolerance [2], adaptive fault-tolerance [5], [6], and extending MapReduce with other fault-tolerance models [7], [8]. All these works evaluate the proposed fault-tolerance solutions in an ad-hoc way. Although some low-level tools exist to test fault-tolerance of Hadoop MapReduce and HDFS [9], there is no principled way to describe faultloads, and to measure reliability and availability of MapReduce clusters. To the best of our knowledge, MRBS is the first dependability benchmark suite for MapReduce. Other works have more specifically studied MapReduce performance benchmarking, such as HiBench [31], MRBench [32], PigMix [33], Hive Performance Benchmarks [34], GridMix3 [35], and SWIM [36].

HiBench consists of eight MapReduce jobs (e.g. *sort*, *word count*, etc.) [31]. The benchmark measures performance in terms of job processing time, MapReduce task throughput, and I/O throughput. While HiBench includes different types of jobs, it does not support concurrent job execution, that is the whole MapReduce cluster is dedicated to a single job at a time, which inhibits cluster consolidation. Thus, it fails to capture different workloads and job arrival rates. Furthermore, HiBench does not consider faultload injection and does not allow the evaluation of MapReduce dependability.

MRBench is a domain-specific benchmark that evaluates business-oriented queries [32]. It uses large datasets and complex MapReduce queries derived from TPC-H [17]. However, as HiBench, MRBench fails to capture job concurrency and arrival rates, workload variations, and it does not evaluate MapReduce dependability in presence of failures.

Similarly, PigMix and Hive Performance Benchmarks use a set of queries to specifically track the performance improvement of respectively Pig and Hive platforms [15]. Pig and Hive run on top of Hadoop MapReduce, the former provides a high-level language for expressing large data analysis, and the latter is a data warehouse system for ad-hoc querying.

GridMix3 takes as input a job trace from a specific workload and emulates synthetic jobs mined from that trace [35]. GridMix3 is able to replay synthetic jobs that generate a comparable job arrival rate and a comparable load on the I/O subsystems as the original jobs in the specific workload did. However, GridMix3 does not capture the processing model and the failure model from the traces. Thus, it fails to reproduce comparable job processing times and failures in the MapReduce cluster.

SWIM is a similar framework that synthesizes specific MapReduce workloads [36]. The framework first samples MapReduce cluster traces, and then executes the synthetic workloads using an existing MapReduce infrastructure to evaluate performance. Here again, the proposed framework does not capture job failures and does not model the dependability of the MapReduce cluster.

VIII. Conclusion

The paper presents MRBS, the first benchmark suite for evaluating the dependability of MapReduce systems. MRBS allows to characterize a faultload, generate it, and inject it in a MapReduce cluster. This covers different

fault types, injected at different rates, which provides a means to analyze the effectiveness of MapReduce fault-tolerance systems in a variety of scenarios. MRBS performs an empirical evaluation of the availability and reliability of such systems, to quantify their dependability levels. It also evaluates the impact of failures on the performance and cost of MapReduce.

MRBS is available as a software framework to help researchers and practitioners to better analyze and evaluate the fault-tolerance of MapReduce systems. The current prototype is provided for Hadoop MapReduce, a popular MapReduce framework available in public clouds. MRBS allows automatic deployment of experiments on cloud infrastructures, which makes it easy to use. The paper describes the use of MRBS to evaluate fault-tolerance capabilities of Hadoop clusters running on Amazon EC2 and a private cloud, and presents two possible uses cases of MRBS. We expect to provide MRBS versions for other cloud infrastructures, and to explore other use cases.

ACKNOWLEDGMENT

This work was partly supported by the Agence Nationale de la Recherche, the French National Agency for Research, under the MyCloud ANR project (http://mycloud.inrialpes.fr/) and the University of Grenoble. Part of the experiments were conducted on the Grid'5000 experimental testbed, developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (http://www.grid5000.fr).

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in USENIX Symp. on Operating Systems Design and Implementation (OSDI), 2004.

[2] Z. Fadika and M. Govindaraju, "LEMO-MR: Low Overhead and Elastic MapReduce Implementation Optimized for Memory and CPU-Intensive Applications," in IEEE Int. Conf. on Cloud Computing Technology and Science (CloudCom), 2010.

[3] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters," in EuroSys, 2011.

[4] M. Eltabakh, Y. Tian, F. Ozcan, R. Gemulla, A. Krettek, and J. McPherson, "CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop," in Int. Conf. on Very Large Data Bases (VLDB), 2011.

[5] H. Jin, X. Yang, X.-H. Sun, and I. Raicu, "ADAPT: Availability-Aware MapReduce Data Placement in Non-Dedicated Distributed Computing Environment," in IEEE Int. Conf. on Distributed Computing Systems (ICDCS), 2012.

[6] H. Lin, X. Ma, J. Archuleta, W.-c. Feng, M. Gardner, and Z. Zhang, "MOON: MapReduce On Opportunistic eNvironments," in ACM Int. Symp. on High Performance Distributed Computing (HPDC), 2010.

[7] A. N. Bessani, V. V. Cogo, M. Correia, P. Costa, M. Pasin, F. Silva, L. Arantes, O. Marin, P. Sens, and J. Sopena, "Making Hadoop MapReduce Byzantine Fault-Tolerant," in IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN), Fast abstract, 2010.

[8] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta, "Making Cloud Intermediate Data Fault-Tolerant," in ACM Symp. on Cloud computing (SoCC), 2010.

[9] "Fault injection framework." http://hadoop.apache.org/hdfs/docs/r0.21.0/faultinject_framework.

[10] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce Online," in USENIX Symp. on Networked Systems Design and Implementation (NSDI), 2010.

[11] H. Liu and D. Orban, "Cloud MapReduce: A MapReduce Implementation on Top of a Cloud Operating System," in IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGRID), 2011.

[12] "MovieLens web site." http://movielens.umn.edu/.

[13] "Wikipedia Dump." http://meta.wikimedia.org/wiki/Data_dumps.

[14] "Genomic research centre." http://www.sanger.ac.uk/.

[15] T. White, Hadoop: The Definitive Guide. O'Reilly Media, 2009. http://hadoop.apache.org/.

[16] A. Sangroya, D. Serrano, and S. Bouchenak, "MRBS: A Comprehensive MapReduce Benchmark Suite," Research Report RR-LIG-024, LIG, Grenoble, France, 2012.

[17] "TPC Benchmark H - Standard Specification." http://www.tpc.org/tpch/.

[18] "Apache Hive." http://hive.apache.org/.

[19] M. C. Schatz, "CloudBurst: Highly Sensitive Read Mapping with MapReduce," Bioinformatics, 2009.

[20] "Apache Mahout." http://mahout.apache.org.

[21] "20 Newsgroups." http://people.csail.mit.edu/jrennie/20Newsgroups/.

[22] "Amazon Elastic Compute Cloud (Amazon EC2)." http://aws.amazon.com/ec2/.

[23] "OpenStack." http://www.cloud.com.

[24] J. claude Laprie, "Dependable computing and fault-tolerance: Concepts and terminology," in 25th International Symposium on Fault-Tolerant Computing, 1995.

[25] F. C. et. al., "Grid'5000: A Large Scale and Highly Reconfigurable Grid Experimental Testbed," Int. Journal of High Performance Computing Applications (IJHPCA), 2006.

[26] "TPC-C: an on-line transaction processing benchmark." http://www.tpc.org/tpcc/.

[27] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in ACM Symp. on Cloud Computing (SoCC), 2010.

[28] A. Brown and D. A. Patterson, "Towards Availability Benchmarks: A Case Study of Software RAID Systems," in USENIX Technical Conf., 2000.

[29] M. Vieira and H. Madeira, "A Dependability Benchmark for OLTP Application Environments," in Int. Conf. on Very Large Data Vases (VLDB), 2003.

[30] J. Duraesa and et. al., "Dependability Benchmarking of Web-Servers," in Int. Conf. on Computer Safety, Reliability, and Security (SAFECOMP), 2004.

[31] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis," in IEEE Int. Conf. on Data Engineering Workshops (ICDEW), 2010.

[32] K. Kim, K. Jeon, H. Han, S.-g. Kim, H. Jung, and H. Y. Yeom, "MRBench: A Benchmark for MapReduce Framework," in IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS), 2008.

[33] "PigMix." http://cwiki.apache.org/confluence/display/PIG/

PigMix.
[34] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A Comparison of Approaches to Large-Scale Data Analysis," in ACM SIGMOD In. Conf. on Management of Data (SIGMOD), 2009.
[35] "Gridmix3 Emulating Production Workload for Apache Hadoop." http://developer.yahoo.com/blogs/hadoop/posts/2010/04/gridmix3_emulating_production/.
[36] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The Case for Evaluating MapReduce Performance Using Workload Suites," in IEEE Int. Symp. on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2011.


L I G